TITLE:

NANO-2 : High-level Parallel Programming Language for Multiprocessor System HYPHEN(Software Science and Engineering)

AUTHOR(S):

ARAKI, Keijiro; ARITA, Itsujiro; HIRABARU, Masaki

CITATION:

ARAKI, Keijiro ...[et al]. NANO-2 : High-level Parallel Programming Language for Multiprocessor System HYPHEN(Software Science and Engineering). 数理解析研究所講究録 1985, 547: 35-57

ISSUE DATE:

1985-01

URL:

http://hdl.handle.net/2433/98846

RIGHT:

京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

数理解析研究所講究録
547巻 1985年 35-57

NANO-2 : High-level Parallel Programming Language for Multiprocessor System HYPHEN

Keijiro ARAKI, Itsujiro ARITA* and Masaki HIRABARU

荒木啓二郎, 有田五次郎, 平原正樹

Department of Computer Science and Communication Engineering
Kyushu University
Hakozaki, Higashi-ku, Fukuoka 812, JAPAN

(* Current affiliation : Department of Computer Science
Kyushu Institute of Technology,
Tobata-ku, Kitakyushu 804, JAPAN)

ABSTRACT

This paper describes the design principles and the features of the high-level parallel programming language Nano-2. Nano-2 has been designed to facilitate the construction of parallel programs which run on the highly parallel processing system Hyphen. A Nano-2 program consists of machine dependent programs called SPPs (Self-synchronizing Parallel Programs) and abstract parallel constructs called processes. Nano-2 is also intended to support the reliable programming process by compiler checks and by the information explicitly expressed in the program text.

1. Introduction

The evolution of computer technology has made possible the construction of various kinds of multiprocessor systems. Hyphen C-16[9] is such a multiprocessor system, which consists of sixteen microprocessors, and was constructed in 1982. It is a general purpose parallel processing system, whose component processors are connected by the hierarchical exchanging network. The Hyphen C-16 system is regarded as a memory shared multiprocessor, and has firmware instructions such as 'parallel branch,' 'exchange task,' 'lock' and 'unlock' for parallel execution.

We aim to construct an integrated software system on the Hyphen system, to accumulate experience in parallel programming in a real parallel environment, and to learn about parallel or distributed processing. We design a Hyphen-oriented parallel programming language we call Nano-2 and implement its compiler since programming languages are the bases for computer software.

Nano-2 has been designed to facilitate the construction of parallel programs which run on the Hyphen system. We did not intend merely to add mechanisms for inter-process communication and synchronization to an existing language such as Pascal or C, but decided to design a new parallel programming language to support structured programming, information hiding, readability, etc. A Nano-2 program consists of three kinds of constructs, i.e., process, segment and task. Segments and tasks are used to describe parallel programs dependent on the Hyphen system. A

process encapsulates machine dependent programs consisting of segments and tasks, and serves as an abstract construct which is executed in parallel with other processes.

Nano-2 draws much on Euclid[8] for its features. Especially, import/export declarations are used to control the visibility of names explicitly as in Euclid. Those declarations are expected to be helpful for understanding programs.

Concurrent Euclid[6] is a programming language based on Euclid, and supports monitors for concurrency. It is intended for writing system software such as operating systems and compilers. Although Nano-2 is also based on Euclid, it is intended for writing distributed programs which run on the multiprocessor system Hyphen.

In this paper we describe design principles and main features of the Hyphen oriented high-level parallel programming language Nano-2. In section 2 we give an overview of the Hyphen system. In section 3 we discuss the design principles of the language Nano-2. In section 4 we briefly describe the program structures and parallel features of Nano-2. In section 5 we present an example of a parallel program written in Nano-2. Concluding remarks are given in section 6.

2. Overview of Hyphen System

Hyphen C-16[9] is a prototype machine consisting of sixteen microprocessors. Fig.1 shows its system configuration. Enclosed by the dotted line is the hierarchical exchanging network, where

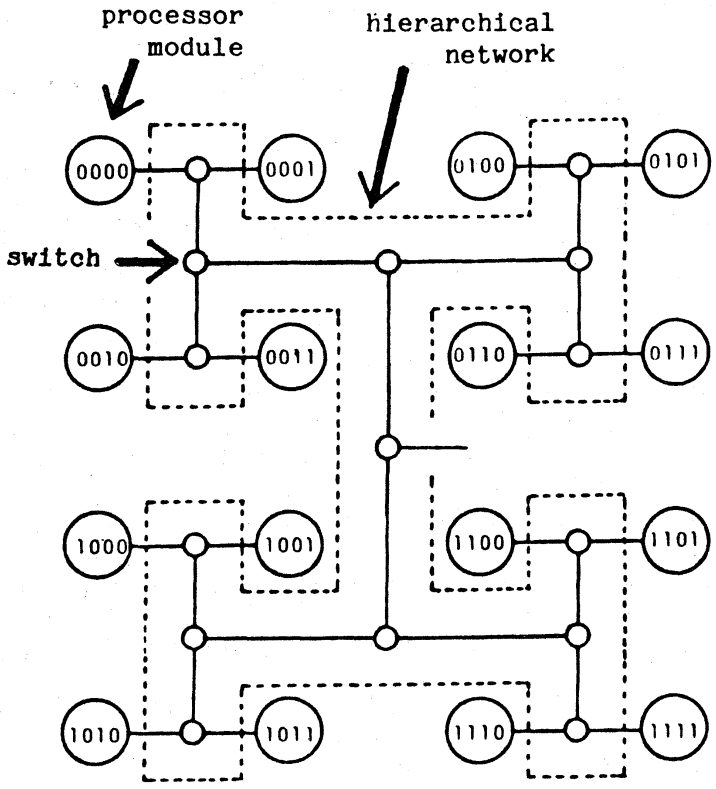


Fig.1 System Configuration of Hyphen C-16

small circles are bidirectional bus switches. The network is a binary tree on which processor modules are connected as leaves (large circles in Fig.1). Though the Hyphen system has a hierarchical structure, all the processor modules are uniformly terminal nodes. Each processor module consists of a processor unit and a memory unit, and has its own address which specifies the module uniquely in the Hyphen system. Table 1 shows the detail of a processor module of the current Hyphen C-16 system.

Table 1. Processor Module

CPU	Z-80A (clock 4MHz)	
Memory	ROM	4KB
	RAM	12KB
Communication Interface	Serial I/O	2 channels
	Parallel I/O	1 channel

A processor unit can access a memory on another processor module via the exchanging network by specifying the address of the memory which is uniquely determined by means of the address of its processor module. Although the memories are distributed among the processor modules, they form a single memory space as a whole. Therefore the Hyphen system is thought of as a memory shared multiprocessor system.

A processor unit executes instructions fetched from its own memory unit, whereas it can access memories of the other modules. A processor unit can activate a program running on another processor module by a 'parallel branch' instruction. Each processor module has a FIFO (First In First Out) queue, where the entry address of the activated program is placed. When the activated processor module executes an 'exchange task' instruction, it ceases to execute the current program, and then removes the head element from the FIFO queue and starts to execute the new program specified by the removed address. If the FIFO queue is empty, the processor module waits for an entry

address newly activated by a 'parallel branch' instruction which another module executes.

We can describe a parallel program which runs on the Hyphen system with such instructions as 'parallel branch' and 'exchange task.' These instructions on the FIFO queue are used for synchronization between processor modules. Arita[2] proposed the notion of SPP(Self-synchronizing Parallel Program) to describe a parallel program by means of the synchronization mechanism with the FIFO queue.

Fig.2 shows a simple example of SPP which calculates the sum of 1 through 4 in parallel. Two processor modules P1 and P2 are used for this calculation. Processor module P1 calculates a subtotal of 1 and 2, and finally makes the grand total. Processor module P2 calculates the other subtotal of 3 and 4. The subtotals are to be calculated in parallel on P1 and P2. See

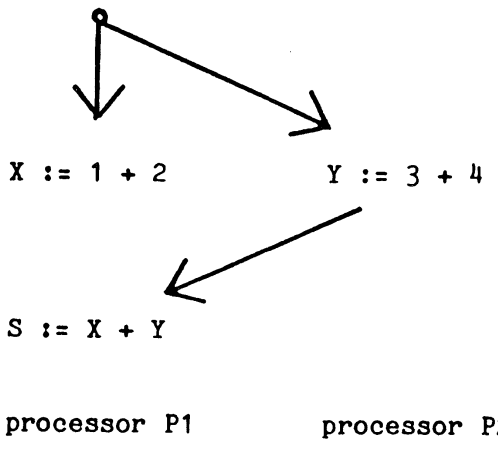


Fig.2 Simple Example of SPP which calculates the sum of 1, 2, 3 and 4 in parallel

also Fig.5 in reading the following explanation of Fig.2. The reader is referred to Arita[2] for the detailed explanations and notations of SPP.

First of all, P1 executes a 'parallel branch' instruction to P2 and continues on to add 1 and 2. P1 executes an 'exchange task' instruction after finishing this addition. P2 is activated by P1 and calculates the subtotal of 3 and 4 in parallel with the addition of 1 and 2 on P1. After finishing the calculation, P2 executes 'parallel branch' to P1. If P1 has already finished the addition of 1 and 2, P1 starts to execute the assignment statement S:=X+Y on being activated by the 'parallel branch' from P2. If P1 has not yet finished the addition, the request for the activation of S:=X+Y is placed in P1's FIFO queue and its execution is deferred until the execution of 'exchange task' after finishing the addition of 1 and 2.

Here we must remark the SPP shown in Fig.2 specifies the allocation of program pieces on the processor modules. The program pieces on the same column are allocated to the same processor module.

Since the Hyphen system has a recurrent structure connected by communication lines and switches as shown in Fig.1, it provides a great extensibility and flexibility. A processor unit in the Hyphen system is a general purpose microprocessor (the current system Hyphen C-16 uses Z-80A processors). Thus we can of course implement other synchronizing algorithms without using FIFO queues.

3. Design Principles of Nano-2

We have designed Nano-2 to write parallel programs which run on the Hyphen system. We took the following considerations into account in designing Nano-2.

(1) Nano-2 is a Hyphen oriented parallel programming language. We intended to be able to describe SPPs in Nano-2. Thus Nano-2 contains machine dependent features concerning 'parallel branch,' 'exchange task' and 'lock/unlock.' Nano-2 does not support generalized parallel or concurrent features such as monitors[5] and rendezvous[1]. We build up those features by using machine dependent parallel operations in Nano-2.

(2) Nano-2 is a high-level programming language, though it contains machine dependent features. Isolation or encapsulation of machine dependencies is effective for software maintenance. In Nano-2, a process serves as an abstract construct which is invoked and run in parallel with the invoker. A process itself is also a parallel construct consisting of SPPs and some other processes. Thus a Nano-2 program is structured hierarchically, but does not have Pascal-like nesting block structures. Of course Nano-2 provides some of structured statements and data types other high-level languages provide.

(3) Nano-2 inherits explicit control over name visibility from the programming language Euclid[8]. Names must be explicitly imported (or exported) via import (or export) declarations. This kind of information expressed explicitly in the program text is valuable for understanding and maintenance of the program[4,7].

(4) Nano-2 supports reliable programming. The compiler will do very stringent checks of types, visibility, and accessibility. These checks help to eliminate many common programming errors[7,10].

(5) We can separately compile some kinds of program components. Libraries of software components improve software productivity.

Although Nano-2 draws much on Euclid, we did not take account of verification of Nano-2 programs. Neither did we take account of ease of implementing a compiler, though we began to design and implement a Nano-2 compiler.

4. Language Features of Nano-2

4.1 Program structure and scope

A Nano-2 program consists of one or more processes. A process is an abstract parallel construct and consists of SPPs and other processes. Thus a Nano-2 program is naturally structured hierarchically.

SPPs are machine dependent parallel programs, and they are described in terms of tasks and segments in Nano-2. SPPs which perform some function are enclosed into a process as an abstract construct. Only entries of the process can be seen from outside.

Fig.3 shows the hierarchical structure of a Nano-2 program consisting of two processes P and R. Process P consists of two processes (Q1 and Q2) and two segments (S1 and S2). Segment S1 is made of two tasks (T1 and T2), while segment S2 is made of

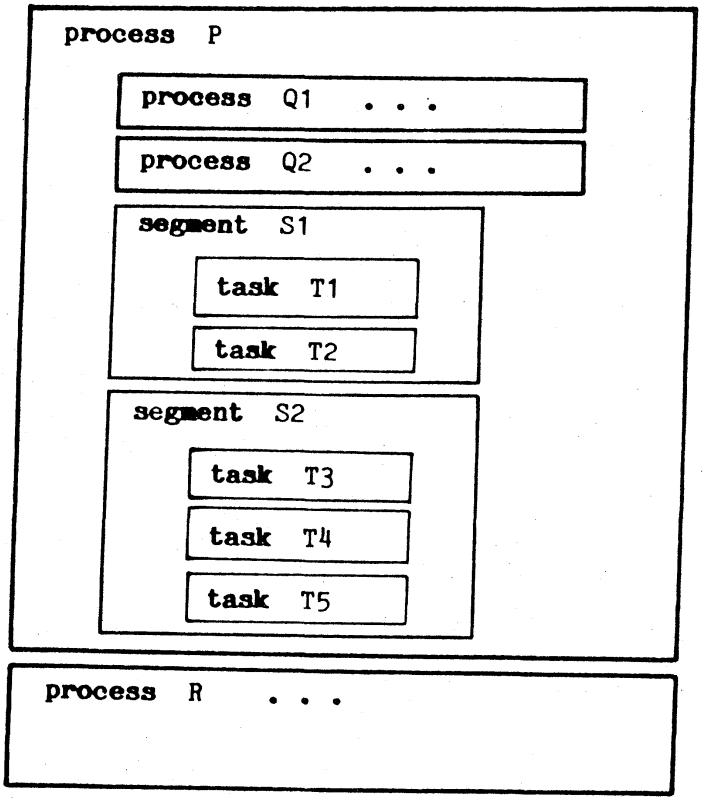


Fig.3 Hierarchical Structure of a Nano-2 Program

three (T3, T4 and T5). SPPs described in section 2 are represented in terms of segments and tasks. A segment is a unit for allocation, i.e., all the tasks included in the segment must be allocated on the same processor module. A task specifies a sequence of statements that are executed continuously. Once a processor module starts to execute one of its tasks, it continues to execute the task until it reaches the end of the task or executes an exit statement. Then the processor module executes an 'exchange task' instruction to start another task.

We may write a Nano-2 program for the SPP shown in Fig.2 as Fig.4, whose structure is shown in Fig.5.

```
process add1234;
  exports entry S12.T12;

  segment S12;
    imports S34;
    exports entry T12;
    var X, S : integer;

    task T12;
      imports S34, T1234, var X;
      begin
        parado S34.T34(T1234);
        X := 1 + 2;
      end T12;

    task T1234;
      imports S34, X, var S;
      begin
        S := X + S34.Y;
      end T1234;

  end S12;

  segment S34;
    exports entry T34, Y;
    var Y : integer;

    task T34(entry e);
      imports var Y;
        begin
          Y := 3 + 4;
          parado e;
        end T34;

  end S34;

end add1234;
```

Fig.4 Nano-2 Program for SPP in Fig.2

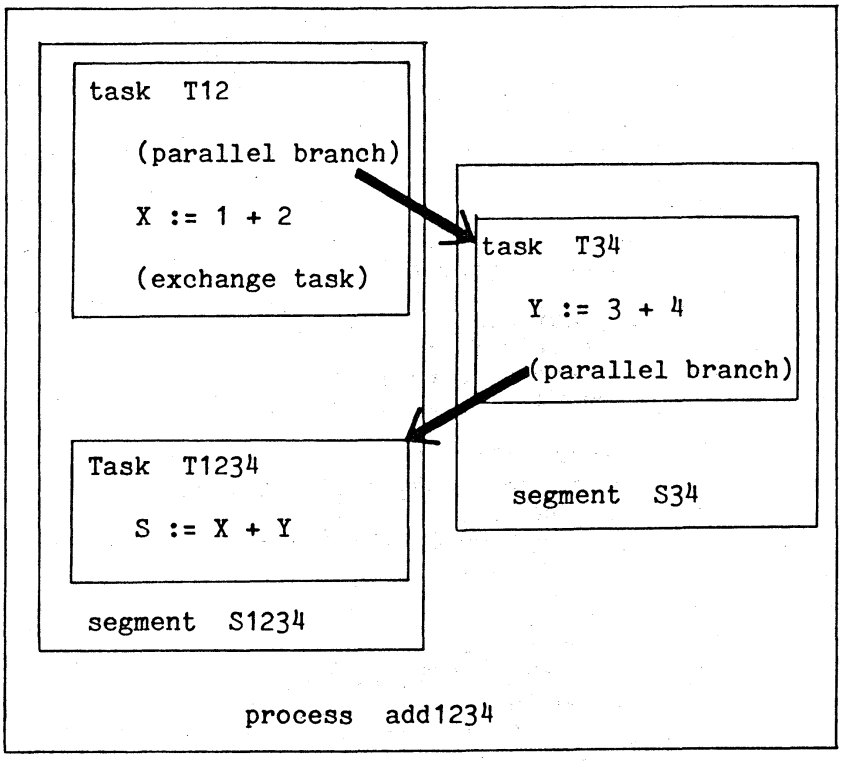


Fig.5 Program Structure of SPP in Fig.4

A Nano-2 program also specifies the static configuration of the program components on the Hyphen system. No processes, segments or tasks are dynamically created during the execution of a Nano-2 program.

A construct in Nano-2 has a closed scope. Visibility of names is controlled explicitly almost in the same way as in Euclid. A name is visible in the scope in which it is declared. If it is to be visible in contained scopes then it must be explicitly imported into those scopes via an import declaration. Names declared in a construct are visible outside of the construct if and only if they are explicitly exported from the construct into the enclosing construct by an export declaration.

We require a Nano-2 programmer to express thoroughly the interface between constructs via import/export declarations. An export declaration corresponds to the service specification offered to others. An import declaration shows the dependence on others outside. These declarations help greatly in understanding a program, and are thought of as documents expressed in the program text.

The behavior of parallel programs could be much harder to understand than that of sequential programs. Programming in an unrestricted manner will make parallel programs more and more complicated. In Nano-2, parallel programs are constructed of processes, segments and tasks, interfaces among which must be expressed in terms of import/export declarations explicitly. Such a constrained programming manner is expected to be effective to make programs reliable and easy to understand.

Finally, the textual order of constructs contained in the same construct does not affect the scope or visibility of the contained constructs. Therefore in Nano-2 the equivalence holds between the two program structures shown in Fig.6, while it does not in Pascal.

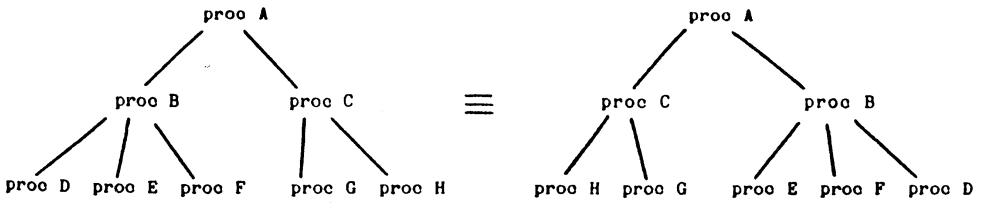


Fig.6 Equivalent Hierarchical Structures

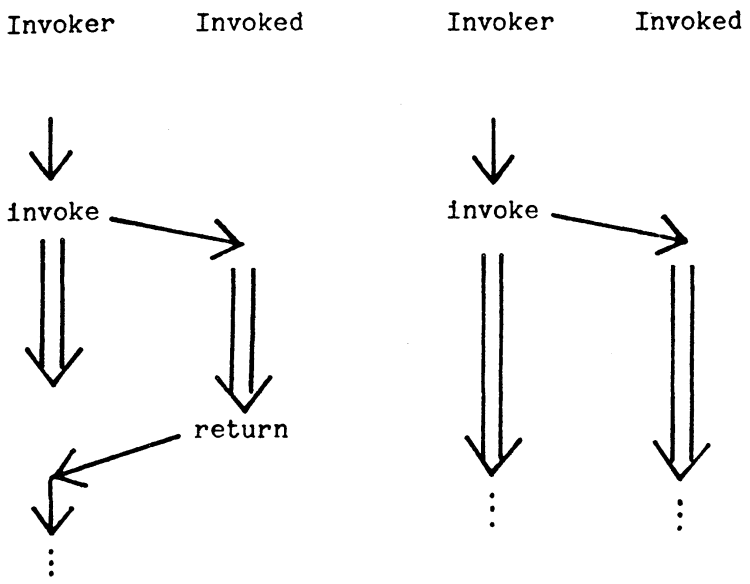
4.2 Entity and allocation

Tasks in the same segment are allocated on a single processor module in the Hyphen system. Data, procedures and functions which are declared in a segment or are imported from the enclosing process to the segment are allocated on the same processor module as the tasks contained in the segment.

Procedures, functions, variables, and tasks within a segment have their own entities. Procedures, functions and data declared directly in a process are not yet allocated on processor modules, thus do not have their entities. They may be considered as the templates for the entities allocated on processor modules. The entities for them are allocated when they are imported into some segments. This facility is prepared in order to describe a procedure or function which is used commonly and independently on more than two processor modules. As examples, we mention library procedures/functions and predefined procedures/ functions. Each of them is defined somewhere only once, and its object code is loaded on every processor module that uses it.

4.3 Interaction between parallel constructs

Processes may have parameters and are invoked in two ways (see Fig.7). Fig.7(a) shows the case in which a return from the invoked process is required. This can be thought of as a remote procedure call[3] with parallel execution between the caller and the called. This may also be thought of as fork-join with



(a) call-return scheme (**paracall** is used for this invocation.)

(b) no-return scheme (**parado** is used for this invocation.)

Fig.7 Control Flow in Process Invocation

parameters. On the other hand Fig.7(b) shows the case in which no return is required. The invoking process and the invoked process may execute in parallel, and they need not worry about the return. This can be thought of as a fork statement with parameters.

The invocation of another process is realized by means of the 'parallel branch' instruction as described in section 2. Thus the entry address of the invoked process is queued. The execution is deferred until the entry address is removed from the FIFO queue. The return from the invoked process is realized in the same way as the invocation. A task invocation within an SPP is treated in just the same way as a process invocation.

Parameters of process invocation are used for inter-process communication as in the remote procedure call. A task in an SPP may also have parameters, which are used for inter-task communication. Besides, a task can communicate with other tasks via shared variables allocated on certain segments.

5. Example

As an example Fig.8 shows a Nano-2 program for 'Dining Philosophers.' The program consists of six processes: five for philosophers and one for the dining room. Fig.9 shows the schematic structure of this program. There are five segments "s1" ... "s5" in the "Dining_room" process. Each of them works as a waiter who is devoted to only one guest. A philosopher who wants to eat spaghetti requests two forks of the "manager" segment in the "Dining_room" process via the "req" entry of one of the waiter segments "s1" ... "s5". The philosopher who requests is blocked until the two forks are available. The "start" task activates the waiting philosopher, and then he/she starts to eat. After eating spaghetti for a while, the philosopher releases the forks and then starts to think.

The "Dining_room" process interacts with the philosophers only via the parameters in process calls, and is thus a server process independent of the users (i.e., philosophers).

The reader may wonder why the part (A) in the program is needed. Suppose a philosopher requests the forks again soon after he/she releases them. If none of his/her neighboring

```
process Dining_room;
  exports entry s1.req(entry r),
          entry s2.req(entry r),
          entry s3.req(entry r),
          entry s4.req(entry r),
          entry s5.req(entry r),
          entry release.e(i:1..5);

  segment release;
    imports manager;
    exports entry e(i:1..5);

    task e(i:1..5);
      imports manager;
      begin
        parado manager.rel(i);
      end e;

  end release;

  segment s1;
    imports manager;
    exports entry req(entry r),
            entry start;

    task req(entry r);
      imports manager;
      exports entry r;
      begin
        parado manager.request(1);
      end req;

    task start;
      imports req;
      begin
        return req.r;
      end start;

  end s1;


  segment s2;
     .
     .
     .
  end s5;
```

Fig.8 Nano-2 Program for Dining Philosophers

```
segment manager;
  imports s1, s2, s3, s4, s5;
  exports entry rel(i:1..5),
          entry request(i:1..5);
  var fork : array 1..5 of boolean
        := (true, true, true, true, true);
  var waiting : array 1..5 of boolean
        := (false, false, false, false, false);
  const left : array 1..5 of 1..5
        := (5, 1, 2, 3, 4);
  const right : array 1..5 of 1..5
        := (2, 3, 4, 5, 1);

  procedure start_eating(i:1..5);
    imports s1, s2, s3, s4, s5;
    begin
      if i = 1 then parado s1.start;
      elseif i = 2 then parado s2.start;
      elseif i = 3 then parado s3.start;
      elseif i = 4 then parado s4.start;
      elseif i = 5 then parado s5.start;
      end if;
    end start_eating;

  task rel(i:1..5);
    imports var fork,
            var waiting,
            left, right,
            start_eating;
    begin
      if fork(left(i))
           and waiting(left(i))
        then fork(left(i)) := false;
             waiting(left(i)) := false;
             start_eating(left(i));
        else fork(i) := true;
      end if;
      if fork(right(right(i)))
           and waiting(right(i))
        then fork(right(right(i))) := false;
             waiting(right(i)) := false;
             start_eating(right(i));
        else fork(right(i)) :=true;
      end if;
      if fork(i) and fork(right(i))
           and waiting(i)
        then fork(i) := false;
             fork(right(i)) := false;
             waiting(i) := false;
             start_eating(i);
      end if;
    end rel;
```

(A)

Fig.8 Nano-2 Program for Dining Philosophers (Continued)

```
      task request(i:1..5);
        imports var fork,
                var waiting,
                right,
                start_eating;
        begin
          if fork(i) and fork(right(i))
            then fork(i) := false;
                 fork(right(i)) := false;
                 start_eating(i);
            else wating(i) := true;
          end if;
        end request;

    end manager;

  end Dining_room;


  process philosopher1;
    imports Dining_room;
    exports s.init;

    segment s;
      imports Dining_room;

      task init;
        imports Dining_room,
                eat_and_think;
        begin
          I_am_thinking;
(B) ----  paracall
            Dining_room.s1.req(eat_and_think);
        end init;

      task eat_and_think;
        imports Dining_room;
        begin
          I_am_eating;
(C) ----  parado Dining_room.release.e(1);
          I_am_thinking;
          paracall
            Dining_room.s1.req(eat_and_think);
        end eat_and_think;

    end s;

  end philosopher1;


  process philosopher2;
      .
      .
      .
  end philosopher5;
```

Fig.8 Nano-2 Program for Dining Philosopfers (Continued)

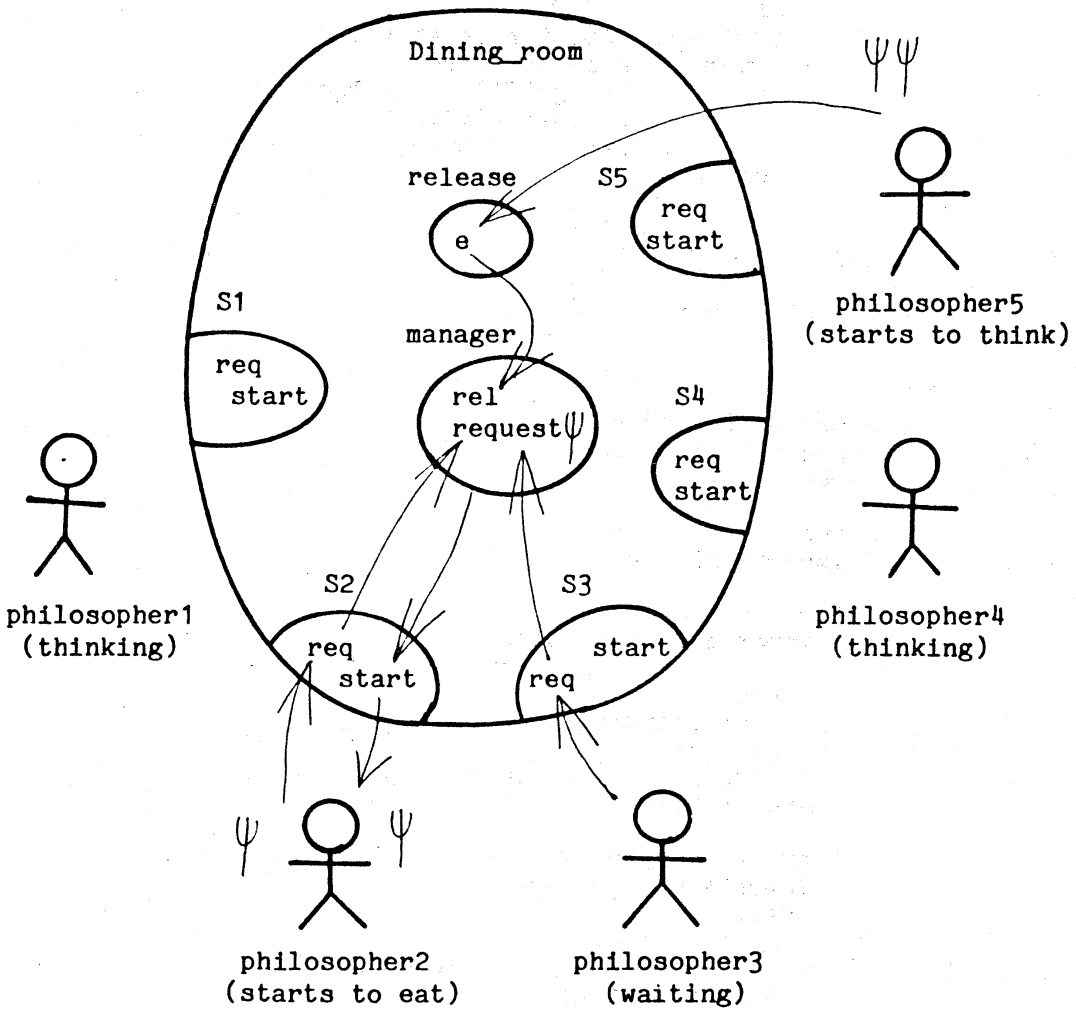


Fig.9 Program Structure for Dining Philosophers
(This figuar shows a snapshot at a particular moment.)

philosophers wants to eat spaghetti, then he/she may use the forks and eat spaghetti immediately. There is no problem in case the "manager" segment deals with the request after the release. But there is no guarantee that the above case always occurs. As the segments in the "Dining_room" process may be allocated on different processor modules, the "manager" segment may deal with the release after the request.

When a philosopher requests two forks, he/she must wait for some reply from the dining room until the forks are available for his/her use. Thus a call-return scheme is used (see (B) in Fig.8). On the other hand, when a philosopher release the forks, he/she need not wait for any reply from the dining room. Thus a no-return scheme is used (see (C) in Fig.8).

6. Conclusion

We have described Nano-2: Hyphen oriented high-level parallel programming language. We regard the parallel computer Hyphen and the language Nano-2 as mere tools for investigation of parallel programming or distributed processing. We have a primary object in studying parallelism from the practical viewpoint.

In this paper we have explained only the conceptions of Nano-2. It was considered most how to make a compromise between the machine dependency and the abstraction of the language. We need to investigate whether our design is proper for our aim by writing programs in various fields. Of course there may be

better decisions in designing the language. There will be some revisions of the language Nano-2.

We may regard Nano-2 as a system implementation language. We implement in Nano-2 and provide some basic facilities for a particular application. Mechanisms for coroutines, monitors, pipelining, data abstraction, and so on can be implemented in Nano-2 by means of operations on FIFO queues.

A compiler for Nano-2 has begun to be implemented. After producing it, we will gain the experience in using Nano-2 on the Hyphen C-16 system. The experience would be valuable not only in revising Nano-2 but in learning parallel programming.

Acknowledgements

We wish to express our gratitude to Prof. T.H. Merrett of McGill University for many helpful suggestions made in the improvement of the manuscript.

References

[1] Ada Programming Language (ANSI/MIL-STD-1815A), United States Government, Department of Defense, Ada Joint Program Office, 1983.

[2] Arita,I.: On a Parallel Program with Synchronizing Mechanism Using FIFO Queue (I) : Self Synchronizing Parallel Program, Trans. IPSJ, Vol.24, No.2, pp.221-229, 1983. (in Japanese)

[3] Brinch Hansen,P.: Distributed Processes: A Concurrent

Programming Concept, Commun. ACM, Vol.21, No.11, pp.934-941, 1978.

[4] des Rivieres,J. and Spencer,H.: Readability and Writability in Euclid, SIGPLAN Notices, Vol.13, No.3, pp.49-56, 1978.

[5] Hoare,C.A.R.: Monitors: An Operating System Structuring Concept, Commun. ACM, Vol.17, No.10, pp.549-557, 1974.

[6] Holt,R.: Concurrent Euclid, the UNIX System, and TUNIS, Addison-Wesley, 1983.

[7] Horning,J.J.: A Case Study in Language Design: Euclid, Lecture Notes in Computer Science, Vol.69, pp.125-132, 1979.

[8] Lampson,B.W., Horning,J.J., London,R.L., Mitchell,J.G. and Popek,G.L.: Report on the Programming Language Euclid, SIGPLAN Notices, Vol.12, No.2, 1977.

[9] Sueyoshi,T., Saisho,K. and Arita,I.: HYPHEN C-16 --- A Prototype of Hierarchical Highly Parallel Processing System, Trans. IPSJ, Vol.25, No.5, 1984. (in Japanese)

[10] Wirth,N.: The Programming Language Pascal, Acta Informatica, Vol.1, pp.35-63, 1971.